# Data Stream Management Systems *

**Sandra Geisler**

**RWTH Aachen University**
**Ahornstrasse 55, 52056 Aachen, Germany**
geisler@dbis.rwth-aachen.de

## Abstract

In many application fields, such as production lines or stock analysis, it is substantial to create and process high amounts of data at high rates. Such continuous data flows with unknown size and end are also called data streams. The processing and analysis of data streams are a challenge for common data management systems as they have to operate and deliver results in real time. Data Stream Management Systems (DSMS), as an advancement of database management systems, have been implemented to deal with these issues. DSMS have to adapt to the notion of data streams on various levels, such as query languages, processing or optimization. In this chapter we give an overview of the basics of data streams, architecture principles of DSMS and the used query languages. Furthermore, we specifically detail data quality aspects in DSMS as these play an important role for various applications based on data streams. Finally, the chapter also includes a list of research and commercial DSMS and their key properties.

**1998 ACM Subject Classification** H.2.4 Systems

**Keywords and phrases** Data Streams, Data Stream Management, Data Quality, Query Languages

**Digital Object Identifier** 10.4230/DFU.Vol5.10452.275

## 1 Introduction

Today, sensors are ubiquitous devices and they are crucial for a multitude of applications. Especially, tasks like condition monitoring or object tracking often require sensors [76]. Important examples for monitoring applications are weather observation and environment monitoring in general, health monitoring, monitoring of assembly lines in factories, RFID monitoring, or road monitoring. These applications share characteristic properties which are especially challenging for a data management system processing this data. First of all, the sensed data is produced at a very high frequency, often in a bursty manner, which may pose real-time requirements on processing applications and may allow them only one pass over the data. ECG signals, for example, are created at a frequency of usually 250 Hz. Second, sensor data is not only produced rapidly, but also continuously forming a *data stream*. Data streams can be unbounded, i.e., it is not clear, when the stream will end. Data from sensors furthermore can be defective, i.e., it is likely to include errors introduced by the imprecision of measurement techniques (e.g., a vehicle speedometer has an error tolerance of 10% [29]), data may be lost due to transmission failure or failure of the sensor.

Also the mapping of recorded sensor data to a time domain is important to rate the timeliness of the data and to make it interpretable in that dimension. In monitoring

* This work has been supported by the German Federal Ministry of Education and Research (BMBF) under the grant 01BU0915 (Project Cooperative Cars eXtended, http://www.aktiv-online.org/english/aktiv-cocar.html) and by the Research Cluster on Ultra High-Speed Mobile Information and Communication UMIC (http://www.umic.rwth-aachen.de).

Data Exchange, Integration, and Streams. *Dagstuhl Follow-Ups*, Volume 5, ISBN 978-3-939897-61-3.
Editors: Phokion G. Kolaitis, Maurizio Lenzerini, and Nicole Schweikardt; pp. 275–304

Dagstuhl Publishing
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany

applications the most recent data is apparently the most interesting data. This reveals another source of defectiveness of the sensor data – namely disorder of data, which is likely to happen when the protocol used for transmission cannot guarantee to sustain order or network latencies occur [64]. In monitoring applications, data from multiple sources have also to be integrated and analyzed to build a comprehensive picture of a situation. For example, integrating data from several health sensors (such as ECG, temperature, blood pressure) has to be integrated to derive that a patient is in a critical situation. Another challenge is caused by the pure mass of data. Due to limited system resources in terms of space and CPU time, algorithms analyzing the data, e.g. data mining algorithms, cannot store and process the entire data, but they can process the data only once (*the one-pass property*) with the available resources. In some applications, it is also required to combine streaming data with historical or static data from a common database, e.g., when for a monitored road section we want to look up, if it currently contains a construction site. All of these properties are especially common to sensor data on which we will focus in this chapter. There are also other typical applications producing data streams (further termed *stream applications*). Popular examples are stock price analysis or network traffic monitoring, which can produce even millions of samples per second.

To tackle the aforementioned challenges in data stream processing, a specific type of systems called Data Stream Management Systems (DSMS) has evolved. The properties of stream applications discussed before lead to a long list of requirements for DSMS. In contrast to common data management systems and based on the nature of stream applications, DSMS have to react to incoming data and deliver results to listening users frequently [65]. This is also termed as the *DBMS-Active, Human-Passive model* by Carney et al. [19], while the *DBMS-Passive, Human-Active model* is implemented by common Database Management Systems (DBMS). In DSMS a reactive behaviour is realized by continuous queries which are registered by a user in the system once and after that the queries are executed incessantly. But some applications may at the same time require to allow ad hoc user queries [2] or views [39] also. Data in a DSMS must not only be processed and forgotten, but the system also has to react to changes of data items, which may lead to recalculation of already produced results. This requires an appropriate change management in the system [1, 8].

Handling unbounded data streams while having only a limited amount of memory available and being restricted in CPU time for processing the data is one of the main challenges for a data stream management system. The creation of incremental results for critical data processing operations and the application of window operators are only two examples of how these issues are solved in DSMS. As already mentioned, most of the applications pose real-time requirements to the data processing system [65]. This implicitly comprises the requirement to be scalable in terms of data rates, which in turn demands techniques for load balancing and load shedding to be incorporated in a stream system. But the system must be also scalable in terms of queries as some application contexts can get complex and require the introduction of several queries at the same time. Therefore, the demand for and the adaptability to newly registered, updated, or removed queries is obvious [20]. This also brings up the need for multi-query optimization, e.g., by sharing results of operators in an overall query plan. Query plan modification during query processing is not only desirable for query optimization, but also to serve Quality of Service demands under varying system resource availabilities [59].

The imperfectness of data has also to be addressed by a DSMS. Unordered data has to be handled adequately, and may be tolerated in controlled bounds. Means to recognize and rate the quality of the data processed are also crucial to make assumptions about the produced

answers or results [73]. Finally, as Stonebreaker et al. [65] demand, a DSMS also has to have a deterministic behaviour, which outputs predictable and repeatable results to implement fault tolerance and recovery mechanisms. This contrasts with non-deterministic components in a DSMS, such as a component randomly dropping tuples to compensate a high system load [78].

Parallel to DSMS, the terms Event Processing, Complex Event Processing (CEP) or Event Stream Processing have evolved. These terms are referring to a concept which is closely related to the notion of data streams. The corresponding systems are specialized on the processing and analysis of events to identify higher level events, such as predicting a political development from Twitter tweets or detecting a serious condition from vital parameter readings. DSMS have a broader application scope, but CEP applications can also be rebuild with DSMS [28]. Cugola and Margara distinguish DSMS and CEP systems as data processing and event detection systems denoting the different focuses. The focus disparities result, among others, in differences in architectures, data models and languages between CEP systems and DSMS [25]. The focus of this chapter is put on DSMS. For a comparison of CEPs and DSMS we refer to the survey by Cugola and Margara [25].

In this chapter, we will give a brief overview of DSMS and of some of the solutions which address the above requirements. As there are already excellent surveys on the concepts of DSMS, e.g., [11, 38, 22, 78, 39], we will focus on two main aspects and discuss them in more detail: query languages for DSMS and data quality in DSMS.

## 1.1 Running Example – A Traffic Application Scenario

Throughout this chapter, we will use a real-time traffic management application as a running example, namely traffic state estimation based on data from multiple mobile traffic sources. In Car-to-X (C2X) communication vehicles can communicate with other vehicles (Car-to-Car) or with the road infrastructure (Car-to-Infrastructure). The data collected and sent is called *Floating Car Data* (FCD). For example, a vehicle can warn other vehicles behind it when it brakes very hard. We consider two kinds of event-based messages sent out by the vehicles: Emergency Braking Light (EBL) messages, which are created when a vehicle has a high negative acceleration, and Warning Light Announcement messages (WLA). The latter are produced when a vehicle turns on its warning light flashers. These messages contain general information about the current state of the vehicle, such as the speed, the location, or the acceleration.

Example 1 shows the schema of a C2X message stream, which is an example for FCD. Another source of a data stream may be anonymously collected mobile phone position data (also called Floating Phone Data (FPD) analogously to Floating Car Data (FCD)). This can be used to derive further traffic information, such as the speed or the traffic state [33].

▶ **Example 1.** In our case study we receive the CoCar messages sent by the equipped vehicles as a data stream. The simplified schema of the stream `CoCarMessage` is as follows,

$$C2XMessage(TS, AppID, Speed, Acceleration, Latitude, Longitude)$$

where `AppID` represents the message type, such as an emergency brake message, `Longitude` and `Latitude` represent the position of the vehicle and `Speed` and `Acceleration` represent the current values for these attributes of the vehicle. Additionally, the data stream elements contain a timestamp `TS` from a discrete and monotonic time domain.

The idea of traffic state estimation based on C2X messages is simple. We divide roads in a given road network into equal-sized sections, e.g., of 100 meters length, as depicted in

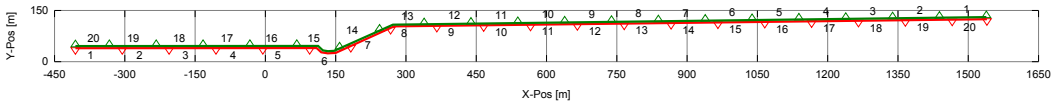



**Figure 1** A road with two directions divided up into sections [35].

Figure 1. We collect the messages and data produced, assign the data to a section based on the position in the message, and aggregate the data for each section and a certain time period (e.g., the last minute). Based on the aggregated data for each section we can then determine the traffic state either based on simple rules (if speed is higher than $x$ and the number of messages is higher than $y$) or using data stream mining techniques.

We realized the described application in the course of the CoCar project[1] and its successor CoCarX. We implemented an architecture based on a DSMS and data stream mining algorithms. The architecture is supposed to derive further traffic data from raw data, such as hazard warnings or the traffic state, and to send it out as traffic information to end-users [35, 34]. The CoCar project investigates the feasibility of traffic applications based on Car-to-X communication via cellular networks (UMTS and its successor LTE) and pWLAN.

## 2 DSMS Architectures

Due to the system requirements for DSMS their architectures differ in several aspects from the traditional relational DBMS architecture. Querying has to be viewed from a different angle as data is pushed into the system and not pulled from the system [20]. In Data Stream Management Systems queries are executed continuously over the data passed to the system, also called *continuous* or *standing queries*. These queries are registered in the system once. Depending on the system, a query can be formulated mainly in two ways: as a declarative expression, mostly done in an SQL-dialect, or as a sequence or graph of data processing operators. Some of the systems provide both possibilities. A declarative query is parsed to a logical query plan, which can be optimized. Similar to DBMS this logical query is afterwards translated into a physical query execution plan (QEP). The query execution plan contains the calls to the implementation of the operators. Besides of the actual physical operators, query execution plans include also *queues* for buffering input and output for the operators. A further support element in QEPs are *synopsis structures*. DSMS may provide specific synopsis algorithms and data structures which are required, when an operator, e.g., a join, has to store some state to produce results. A synopsis summarizes the stream or a part of the stream. It realizes the trade-off between memory usage and accuracy of the approximation of the stream. Additionally, *load shedders* can be integrated in the plan, which drop tuples on high system loads. In most systems, execution plans of registered queries are combined into one big plan to reuse results of common operators for multiple queries. The physical query plan may be constantly optimized based, e.g., on performance statistics. In Figure 2 the query processing chain is depicted.

In Figure 3 a generic architecture of a DSMS based on [5, 39] is shown. First of all a DSMS typically gets data streams as input. Wrappers are provided, which can receive raw data from its source, buffer and order it by timestamp (as e.g. implemented by the *Input Manager* in the STREAM system [64]) and convert it to the format of the data stream

[1] `http://www.aktiv-online.org/english/aktiv-cocar.html`

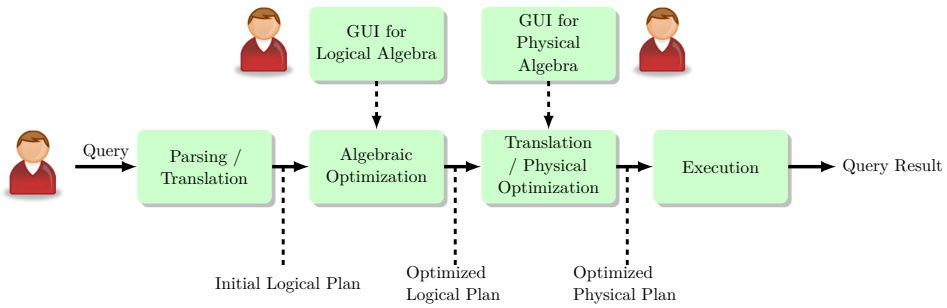



**Figure 2** Illustration of the Query Processing Chain in DSMS as given in [50].

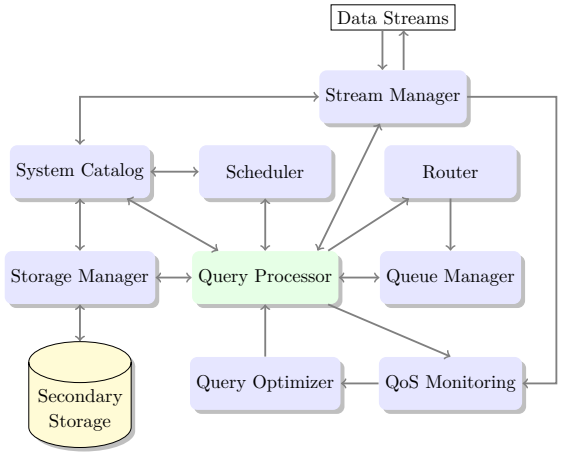



**Figure 3** An generic architecture of a DSMS based on [5, 39].

management system (the task of the *Stream Manager*). As most systems adopt a relational data model, data stream elements are represented as tuples, which adhere to a relational schema with attributes and values. After reception the tuples are added to the queue of the next operator according to the query execution plan. This can be done e.g. by a *Router* component as implemented in the Aurora system [3]. The management of queues and their corresponding buffers is handled by a *Queue Manager*. The Queue Manager can also be used to swap data from the queues to a secondary storage, if memory resources get scarce. To enable access to data stored on disk many systems employ a *Storage Manager* which handles access to *secondary storage*. This is used, when persistent data is combined with data from stream sources, when data is archived, or swapped to disk. Also it is required when loading meta-information about, inter alia, queries, query plans, streams, inputs, and outputs. These are held in a system catalog in secondary storage.

While the queue implementation decides which element is processed next, a *Scheduler* determines which operator is executed next. The Scheduler interacts closely with the *Query Processor* which finally executes the operators. Many systems also include some kind of *Monitor* which gathers statistics about performance, operator output rate, or output delay. These statistics can be used to optimize the system execution in several ways. The scheduler strategy can be influenced, e.g., prioritizing specific subplans. Furthermore, the throughput of a system can be increased by a *Load Shedder*, i.e., stream elements selected by a *sampling* method are dropped. The Load Shedder can be a part of a Query Optimizer, a single component, or part of the query execution plan. Furthermore, the statistics can be used to

reoptimize the current query execution plan and reorder the operators. For this purpose a *Query Optimizer* can be included. In the Telegraph system [20] subsets of operators are build which are commutative. The operators of each subset are connected to a component called Eddy. Each Eddy routes the incoming tuples to the operators connected to it based on optimization statistics. Each tuple has to log, which operator has already processed it successfully and when all attached operators processed it, it is routed to the next part of the plan or output to recipients [20]. DBMS and DSMS share some of their query optimization goals – both try to minimize computational costs, memory usage and size of intermediate results stored in main memory. But obviously, the priorities for these goals are different for the two system types. DBMS mainly try to reduce the costs of disc accesses [27], while DSMS mainly have to reduce memory usage and computation time to be fast enough. Of course, these different goals stem from the different data handling strategies (permanent storage vs. real-time processing). In a DSMS a query is also not only optimized before execution, but it is adaptively optimized during its run time. This enables the system to react to changes of input streams and system and network conditions.

An interesting work to research the different architectural components of a DSMS is the Odysseus [17, 32] framework. Odysseus allows to create customized DSMS for which it provides basic architectural components while offering variation points for each of the components. At these variation points custom modifications such as the integration of a new data model or the inclusion of new algebraic and physical operators or new optimization rules for logical query plans are possible.

Now that general concepts of DSMS architectures for query management and processing have been introduced, the user interface to access the data, namely the principles of query languages in DSMS are discussed in the next section.

## 3 Query Languages in DSMS

In principle, two main types of query languages for DSMS can be distinguished: declarative languages (mostly relational, based on SQL) and imperative languages which offer a set of operators (also called box operators) to be assembled to a data-flow graph using a graphical user interface. The imperative languages often also include operators which represent SQL operations. SQL-based languages are widely used, though SQL has many limitations for querying streams [52]. Systems which include a declarative SQL-based language are, for example, STREAM (Continuous Query Language, CQL) [10], Oracle Event Processing[2], PIPES [50], SASE (Complex Event Language) [43], or StreamMill (Expressive Stream Language, ESL) [72]. Imperative languages are supported for example by the Aurora/Borealis system (SQuAl) [3], or System S/InfoSphere Stream[3] (SPADE/SPL) [31].

Because a stream is potentially unbounded in size, it is neither feasible nor desirable to store the entire stream and analyze it. Some of the operations known from traditional query languages, such as SQL, might wait infinitely long to produce a result, as the operation would have to see the entire stream to generate a result (defined as *blocking operations*) [11]. The missing support of sequence queries, i.e., retrieving sequential data, is one crucial limitation known from relational databases and SQL [52]. One simple yet powerful way is to first extract only a desired portion of the stream and use this portion in the remainder of the query. Therefore, a very important requirement for a DSMS query language is the provision

[2] http://oracle.com/technetwork/middleware/complex-event-processing/overview/index.html
[3] http://ibm.com/software/data/infosphere/streams

of windows [22, 9, 57]. *Windows* are operators that only select a part of the stream according to fixed parameters, such as the size and bounds of the window. Hence, they provide an approximation of the stream, but are at the same time implementing the desired query semantics [11]. A window is updated based on fixed parameters [57] and internal matters of the system (e.g., in principle, a result can be updated whenever a new element arrives or whenever time proceeds) [45]. We will detail the different types and parameters of windows in Section 3.1.5.

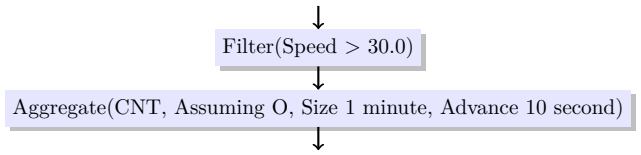
Besides the definition of windows in the language, also an approved set of query operations accompanied by established semantics is beneficial for a DSMS query language [9]. This can be accomplished, e.g., by using a common query language based on relational algebra and its operators for operations on finite tuple sets (commonly called *relations*). The advantage is the reuse of operator implementations and transformations for query optimization [9]. But such an approach also risks to be complicated, because the closure under a consistent mathematical structure, such as bags in relational algebra, is crucial to enable nested queries and algebraic optimization. Cherniack and Zdonik [22] investigated the property of DSMS query languages to be closed under streams. A language is closed under streams if its operators get streams as an input and if the output of the operators are streams as well. They state that most languages provide stream-to-stream operators by either implicitly using operators for windowing and conversion to streams or special stream-to-stream operators. Only CQL provides the possibility to explicitly formulate the conversion of relations to streams by specific relation-to-stream operators. These operators can either add output elements to a result stream when a new element compared to the last time step is in the result set (Istream operator), or when an element has been removed from the query result relation compared to the last time step (Dstream operator), or all elements which are present in the result set in the current time step (Rstream) are added to the stream [10, 9]. But to avoid an inconsistent query formulation producing results not closed under streams, CQL also offers many default query transformations. In fact, Arasu et al. showed in [9] that a stream-only query language (only using stream-to-stream operators) can be build based on the set of CQL operators. We will detail the types of operators used in continuous query languages in Section 3.1.4.

▶ **Example 2.** Suppose we want to monitor the number of speeders in a reduced speed area. We would like to retrieve the number of CoCar messages which have been sent in the last minute and which contain a speed greater than $30km/h$. Additionally, we also want to retrieve an update every 10 seconds. In CQL the following query could be formulated to fulfil our information requirement:

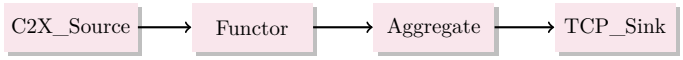
```
SELECT Istream Count(*) FROM
  C2XMessage[Range 1 Minute Slide 10s]
WHERE
  Speed > 30.0
```

**Listing 1** Query 1

The same query can be formulated in an imperative query language by assembling box operators. In Figure 4a a query formulated in Aurora's SQuAl is depicted. The Filter operator is similar to the selection in relational algebra – it retrieves all tuples with speed greater than 30.0. Furthermore, an Aggregate operator is connected to the Filter operator, which can be parametrized with the aggregate function and details for an implicit window integrated in the operator. The SPADE query in Figure 4b is similar – the parametrized Functor operator selects the tuples according to the speed restriction and the Aggregate operator executes the Count operation.




**(a)** Query 1 in SQuAl




**(b)** Query 1 in SPADE

**Figure 4** Query 1 formulated in imperative languages.

To include more powerful and custom functionality, another requirement to DSMS query languages is the extension of the language by custom functions and operators to include more complex actions, such as data mining or custom aggregates [52]. Most of the languages provide a possibility for custom extensions. In the System S by IBM [31] a code generator enables users to create skeleton code for user-defined operators in C++ and add custom code to the operator. The operator is treated the same way as the other operators, e.g., during query optimization. The Aurora system provides a specific operator *Map* in which user-defined functions can be integrated and are executed whenever a new element is received [3]. In the Stream Mill ESL [52, 72] User-defined Aggregates (UDAs) based on concepts from the SQL:99 standard (INITIALIZE, ITERATE, INSERT INTO RETURN, TERMINATE) and the stream are used to implement new operators such as non-blocking aggregates or windows. The UDAs process one tuple at a time and allow for keeping a state over the so far seen stream elements. They can output a result when a specified condition is fulfilled and not only when all the input elements have been seen. In [72] the authors also show how they integrate stream mining algorithms into Stream Mill using UDAs.

## 3.1 Data Models and Semantics of Data Streams

Before we go into detail on continuous queries and operators used in the queries, we have to understand what is exactly meant by the term "data stream". In this section we will also break data streams down to their indivisible components and examine the data models applied in DSMS.

### 3.1.1 Data Models

Depending on the desired application realized with the DSMS and the data sources to support there are different demands on the adopted data model and the semantics of the query language. In the literature, the relational model is very dominant, presumably due to the well-defined semantics, the established set of relational algebra operators, and the very well studied principles of DBMS. Also, from a user's perspective, the step from SQL towards its streaming extensions seems to be quite small. But there are also data sources which do not conform to the flat table concept for discrete data offered by the relational data model [56]. XML data received from, e.g., web services or RSS feeds as well as continuous signal data, object streams, or spatio-temporal data streams demand special treatment. Therefore, a variety of specialized DSMS with differing data models and corresponding query languages have been proposed.

The maintenance and analysis of massive and dynamic graphs is also a challenging problem which can be tackled by data stream technology. There are multiple forms of this data model possible. The stream can consist of, amongst others, a sequence of edges [30], a stream of graphs where each graph is an adjacency matrix or list [67], or weight updates of edges [6]. It may be used in event detection and forecasting e.g. of Twitter streams, for web page linkage or social network analysis. To achieve a stable logical foundation for DSMS Zaniolo presents a data model and query language called Streamlog, which is based on Datalog [77]. While stream sources are represented by fact streams and sinks (or output streams) by the goal of a datalog query, the operators in between are defined by rules.

An interesting generic approach, supporting multiple query languages and data models, is implemented by the Odysseus framework [17]. A logical algebra component converts queries from various language parsers to a logical query plan with generic algebra operators. The semantics of the PIPES system [50] also allows for implementing multiple data models. In this chapter we will concentrate mostly on the classical relational model, though there are also many works on XML stream processing, e.g., using XML-QL [21], XPath [58] or XQuery [18]. In the following we will discuss various representations and semantics of data streams and concepts related to them.

### 3.1.2 Representations and Semantics of Data Streams

A data stream $S$ can be understood as an unbounded multiset of elements, tuples, or events [52, 11, 26, 50]. This means, each tuple can occur more than once in the stream which is denoted by a multiplicity value. Each tuple $(s, \tau) \in U$, where $U$ is the support of the multiset $S$, has to adhere to the schema of $S$. The *support* $U$ of a multiset $S$ is a set which consists of the elements which occur in the multiset $S$ at least once. Multiset and support can be defined as follows [68, 63]:

▶ **Definition 3.** (Multiset and Support of a Multiset) A *multiset* is a tuple $\mathcal{A} = (B, f)$, where $B$ is a set and $f$ is a function $f : B \to \mathbb{N}$, assigning a multiplicity to the tuple (number of occurrencies of the tuple in the stream). The *support* of $\mathcal{A}$ is a set $U$ which is defined as follows:

$$U = \{x \in B | f(x) > 0\},$$

i.e., $U \subseteq B$.

The schema of a stream is constituted of attributes $A_1, \ldots, A_n$. Each tuple contains also an additional timestamp $\tau$ from a discrete and monotonic time domain. Most definitions of data stream semantics do not consider the timestamp as a part of the stream schema [10, 50, 3] and therefore, it is always separately listed in the tuple notation.

A data stream can be formally defined as follows, based on the definition of Arasu et al. [10]:

▶ **Definition 4.** (Data Stream) A data stream $S$ is an unbounded multiset of data stream elements $(s, \tau)$, where $\tau \in T$ is a timestamp attribute with values from a monotonic, infinite time domain $T$ with discrete time units. $s$ is a set of attribute values of attributes $A_1, A_2, \ldots, A_n$ with domains $dom(A_i), 1 \leq i \leq n$, constituting the schema of $S$. A stream starts at a time $\tau_0$. $S(\tau_i)$ denotes the content of the stream $S$ at time $\tau_i$, being $S(\tau_i) = \{< (s_0, \tau_0), m_0 >, < (s_1, \tau_1), m_1 >, \ldots, < (s_i, \tau_i), m_i >\}$.

▶ **Example 5.** The schema of the C2XMsgs stream from Example 1 would be written according to this definition as:

$$C2XMessage(Timestamp, (TS, AppID, Speed, Acceleration, Latitude, Longitude))$$

In this example the timestamp `Timestamp` has been generated by the DSMS and `TS` is the creation timestamp defined by the application. The different kinds of timestamps are detailed in Section 3.2.

Depending on the data model, the attributes $A_1, \ldots A_n$ can contain values of primitive data types, objects or XML data. For example, Krämer and Seeger consider tuples to be drawn from a composite type (in the relational case this is the schema) and its attributes can contain objects [50]. This generic definition allows them to be open to different data models. Gürgen et al. propose a generic relational schema for streams of sensor data of all kinds. Each tuple in a stream consists of several general attributes also denoted as *properties* containing the meta-data of the sensor, an attribute *measurement* carrying the values measured by that sensor and a timestamp, when the measurement has taken place [41]. Their semantics of a stream considers a temporal aspect, i.e., the present, past, and future of a stream is defined. The life time of present data is limited by the size of the corresponding queue in the DSMS and becomes past data when the queue is full [41].

Data streams commonly adhere to an append-only principle, i.e., data once inserted in the stream will not be removed or updated [14]. But to enable updates in the streams, there are also systems which do not only support insertion of tuples, but also updates and deletions. In the Borealis system [1], which is the distributed successor system of Aurora, an extended data stream model has been implemented. The schema of the stream has one or more attributes which have been designated as the key of the stream. This allows for the identification of data stream elements in the stream system for future changes. Furthermore, the tuples include a revision flag which indicates if the tuple is either an insertion, an update, or a deletion. The revision flag is processed by operators in the query plan and if an update or deletion has been detected, former buffered results are reevaluated and also tagged with a corresponding revision flag. Additionally, each tuple can also have information about Quality of Service attached, which will be detailed in Section 4. The internal (physical) representation of streams in STREAM uses the concept of revision flags, too, to denote inserted and outdated tuples, e.g. for a window [10].

In the PIPES [50] and STREAM systems, streams are separated into *base streams* and *derived streams* to denote the origin of the stream. Base streams are produced by an external source and derived streams are produced by internal system operators. Furthermore, in PIPES stream notations are distinguished based on the level in the query processing chain. Streams from external sources are termed *raw streams* and adhere to the attributes plus timestamp notation described above (according which the tuples are ordered in the stream). Streams on the logical or algebraic level are termed *logical streams*. The tuples of a logical stream contain attributes corresponding to the stream schema, a timestamp $\tau$ and a multiplicity value. The multiplicity value indicates how often the tuple occurs at time $\tau$ in the stream, which implements the bag semantics explicitly. Finally, the PIPES system also introduces a *physical stream* notation. The notation is used to represent streams in query execution plans, which include in addition to the attributes a validity time interval with start and end timestamp, similar to the CESAR language [26], which indicates when a tuple is outdated.

While above we have described the representations of streams and tuples, the semantics or denotation of a stream (i.e., the underlying mathematical concept) can be separated from these representations [56]. So far we used the rough semantics of an unbounded multiset for a stream. This rough semantics raises the questions of how tuples are organized in the stream, which tuples are included, and how a stream evolves with the addition of tuples. A stream denoted as a *multiset* or *bag* of elements, allows duplicate tuples in the stream [10, 15, 50].

The denotation as a *set*, i.e., without duplicates, is also possible. Furthermore, a stream could also be interpreted as a sequence of states [56], where a relation transitions from one state to another (on arrival of tuples or progression of time). Maier et al. propose reconstitution functions to formally describe the denotations of streams and operations on these denotations [56]. For example, the insertion of a new tuple in a stream denoted as a bag can be recursively defined by describing what will be the following state, i.e., the "result bag". The reconstitution functions allow to write down the semantics in a more formal way and can then be used to prove properties like the correctness of operators for the corresponding denotation.

### 3.1.3 Inclusion of Persistent Relations

Solely querying of streams is often not sufficient to implement certain applications. Take the traffic state estimation as an example. To improve our assertion about the traffic state we could also integrate information from other streams, such as Floating Phone Data (positions from anonymously tracked phones in vehicles), or from persistent sources such as historical data about the traffic state at the same time last week or last year on the same street. This would involve to join data of streams with other streams and to join streams with persistent data from "static" data sources. In [22] this ability is called *correlation*. Most languages support joins between streams and relations, because it is easy to implement. In principle, each time new tuples arrive in the stream these are joined with the data in the relation and the operator outputs the resulting join tuples. The join between streams is a little bit more complicated. Most languages require at least one stream to be windowed [22], which results in the former situation of joining a stream with a finite relation. Some offer also specific operators for joins without windows.

The use of persistent relations in queries requires them to be represented in the query language. In SQL-based languages these are noted in the same way as streams (it is maybe more correct to say that the streams are represented in the same way as relations). Representations for relations can also be related to the notion of time. A relation at time $\tau$ consists then of a finite, unbounded bag of tuples which is stored in the relation at time $\tau$ [10]. In CQL, a time-related relation is called an *instantaneous relation*, and analogously to streams, *base relations* and *derived relations* are distinguished.

▶ **Example 6.** In this example we want to know for a certain road and section on this road, if and how many road works it currently contains. The schemas of the stream with aggregated information from the C2X messages about the road section and the persistent relation are as follows:

$$AvgC2XMessage(Timestamp, (AvgSpeed, AvgAcceleration, RoadID, SectionID))$$

$$ConstructionSite(SiteID, StartDate, RoadID, SectionID)$$

In CQL we would formulate the query in the following way:

```
SELECT Rstream m.* , Count(c.SiteID) As SiteNo
FROM AvgC2XMessage[Now] AS m, ConstructionSite AS c
WHERE c.RoadID = m.RoadID AND c.SectionID = m.SectionID
```

Note, that CQL is limited in its ability to join streams and relations. Only a `NOW` window on the stream and the Rstream operator can safely be used for joins between streams and relations. Queries with different windows or relation-to-stream operators would usually deliver semantically incorrect results [10].

### 3.1.4 Continuous Queries and Algebraic Operators

Now that we have clarified the main constituents of continuous queries in DSMS, namely, streams, tuples, and relations, we will proceed to explain continuous queries and operators for languages based on the relational algebra. So what is the difference between a continuous query and an ad-hoc query? One goal of Tapestry [70], one of the first data management systems processing continuous data, was to give the user the impression that a query is continuously executed (which is not possible). To achieve this in DSMS, the result of a continuous query at time $\tau$ is equal to the result of the query executed at every time instant before and equal to $\tau$ [70, 10]. That means it takes into account all tuples arrived up to $\tau$. Depending on the language the result of the query can be a stream or a finite set of tuples, e.g., in CQL the result can be either or, while other languages only support streams.

There are three main models how data is processed in a DSMS, i.e., when continuous queries are executed. When a *time-driven model* [45] is used, a query will be updated with progression of time (on every time step of the system). In a *tuple-driven model* a query is evaluated on the arrival of each tuple, unless the query includes some temporal restriction, such as a time-based window [45]. The *event-driven model* allows to define events or triggers on whose firing the query is executed, e.g. in OpenCQ [55]. Of course these could be also temporal events or an amount of tuples seen so far, but these could be also user-defined events, such as a fired alert or an incoming e-mail.

One main problem for operators processing streams is the fact that streams are unbounded. Especially, *blocking operators*, i.e., operators which do not produce a result tuple before they have seen all tuples on their inputs [11], are problematic. The set of these operators comprises aggregations, groupings, but also set operations, such as `NOT IN` or `NOT EXISTS` [52, 40]. For example, if we would like to calculate the average over the speed of the incoming C2X messages, a classical average operation has to wait until the stream of messages ends (but we do not know when the stream ends) to produce the desired result. In contrast, *non-blocking operators* produce results periodically or on arrival of new tuples, i.e., incrementally [51]. *Partially blocking operators* [52] are operators which can produce intermediate results but also a final result in the end. Obviously, a language for continuous queries can then only be based on non-blocking operators [52, 11]. But the set of non-blocking operators is neither in relational algebra nor in SQL sufficient for all expressible relational queries [52]. We discuss the completeness of languages in Section 3.3. Another type of operators which are harmful to continuous query processing are *stateful operators* [56] (in contrast to *stateless operators*). These operators, e.g., joins, require to store a state for their operation, which for streams is unbounded in size. Hence, the remedy to these problems is to approximate processing the stream as a whole as good as possible. One simple yet powerful approach is the partitioning of the stream into small portions, so-called *windows*. Each window is a finite bag of tuples and can be processed also by the common relational blocking operators. A second possibility is to provide incremental implementations of these operators, which are able to update the result with new tuples and output "intermediate" results. Finally, an approximation of the stream in form of a summary or *synopsis* can be used to operate on. In the following we will detail window operators and their semantics.

### 3.1.5 Windows

In continuous query languages based on SQL, windows are a crucial extension to the algebraic set of operators. It depends on the language which types of windows are supported. A window is always built according to some ordered *windowing attribute* which determines the order of

elements included in the window [57, 56]. The type of window, i.e., how it is determined which elements are valid in the current window, according to [57] can be described by its measurement unit, the edge shift, and the progression step. The *measurement unit* can be either a number of $x$ time units (*time-based window*) or tuples (*tuple-based window*) declaring that the elements with timestamps within the last $x$ time units or the last $x$ elements are valid for the window at the point in time of the query. We define a time-based window of size $l$ similar to the definition in [57] as follows:

▶ **Definition 7.** (Time-based Window) A *time-based window* $W_{l_T}$ (with window size $l_T \in \mathbb{T}$) over a stream $S$ at time $\tau_i \in T$ is a finite multiset of stream elements with

$$W_{l_T}(S(\tau_i)) = \{< (s_k, \tau_k), m_k > \mid (s_k, \tau_k) \in U, \tau_i - l_T \leq \tau_k \leq \tau_i, \tau_k \geq \tau_0\},$$

where $m_k$ is the multiplicity of the tuple in the subset and U is the support of stream $S(\tau_i)$.

We assume, that the stream elements are ordered by timestamp $\tau$ before a window operator is applied. The definition for a tuple-based window of size $l_N$ is similar:

▶ **Definition 8.** (Tuple-based Window) Let $S(\tau_i) = \{< (s_0, \tau_0), m_0 >, \ldots, < (s_n, \tau_n), m_n >\}$, $\tau_i \geq \tau_n \geq \tau_0 \wedge \tau_j \geq \tau_{j-1}\ \forall j \in \{1, \ldots, n\}$, be the content of stream $S$ at time $\tau_i$. Then a *tuple-based window* $W_{l_N}$ (with window size $l_N \in \mathbb{N}$) over stream $S$ at time $\tau_i \in T$ is a finite multiset of stream elements with

$$\begin{aligned} W_{l_N}(S(\tau_i)) = \{< (s_k, \tau_k), m_k > \quad \mid \quad & (s_k, \tau_k) \in U, j \leq k \leq i, \\ & \exists m'_j, m''_j.\ m'_j \geq 0, m''_j > 0, \\ & m_j = m'_j + m''_j, \quad m''_j + \sum_{\ell=j+1}^{i} m_\ell = l_N\}. \end{aligned}$$

where $(s_n, \tau_n) \in U,\ \forall \tau_j \in U\ \tau_n > \tau_j$, and U is the support of stream $S(\tau_i)$.

This means, that we go backwards in the stream from time $\tau_i$ on and search for the last point time $\tau_n$ at which elements arrived. We now collect exactly $N$ tuples going backwards in the stream. We assume, that the last tuple which (partially) fits into the window is $< (s_j, \tau_j), m_j >$. Then only the multiplicity portion $m''_j$ which still fits into the window will be added to the window.

In the Aurora system [78, 3] value-based windows as a form of generalization of time-based windows are presented. These windows implement the idea of having a different windowing attribute instead of a timestamp – the attribute just has to be ordered. Furthermore, the value-based window should return only those tuples whose value of the particular attribute is within a specific interval (hence, value-based windows). A similar concept are predicate-windows [37] suited for systems which work with negative and positive tuples. A correlation attribute identifying the data in the tuple and a predicate condition are defined for the window, which enables to filter the tuples according to that predicate. The filtering may result in negative tuples, when the predicate has been fulfilled by tuples for the same correlation attribute value before but is not for the present query evaluation. *Partitioned windows* [53, 10] are applicable to time- or tuple-based windows and follow the idea of dividing the stream into substreams based on filter conditions and of windowing them separately. Afterwards, the windows of the substreams are unioned to one result stream [3].

The *edge shift* of a window describes the motion of the upper and lower bounds of the window. Each of them can either be fixed or moving with the stream. For example, in the

most common variant, the *sliding window*, both bounds move, while for a *landmark window* one bound is fixed and one is moving.

Finally, the *progression step* or *periodicity* defines the intervals between two subsequent movements of a window. This again can either be *time-based* or *tuple-based*, e.g., the window can move every 10 seconds or after every 100 arrived tuples. When the contents of windows in each progression step are non-overlapping, this is termed a *tumbling window*, i.e., size and sliding step have an equal number of units. Windows can also be *punctuation-based*. A notification tuple sent with the stream indicates the window operator that it should evaluate. We will explain punctuations in Section 3.2.

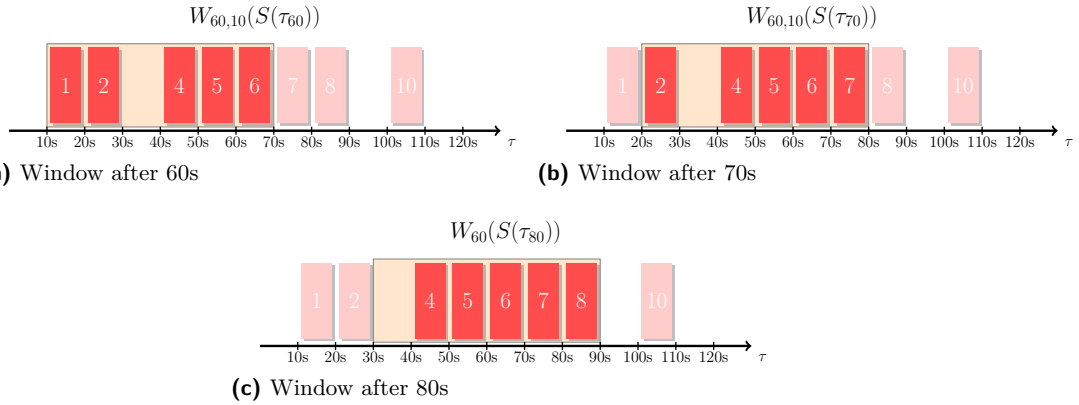
In the following, one of the most commonly used forms of a moving window, a sliding time-based window, is defined as:

▶ **Definition 9.** A *sliding time-based window* with window size $l_T \in \mathbb{T}$ and slide value $v \in \mathbb{T}$ over a stream $S$ at time $\tau_i \in T$ is a finite multiset of stream elements with

$$W_{l_T,v}(S(\tau_i)) = \{< (s_k, \tau_k), m_k > \quad | \quad (s_k, \tau_k) \in U, \exists j \in \mathbb{N} : \tau_0 + j \cdot v \leq \tau_i, \\ \tau_0 + (j+1) \cdot v > \tau_i, \tau_i \geq \tau_0 + l_T, \\ \tau_0 + j \cdot v - l_T \leq \tau_k \leq \tau_0 + j \cdot v, \tau_k \geq \tau_0\}.$$

where U is the support of stream $S(\tau_i)$

▶ **Example 10.** In our traffic example we would like to retrieve the number of C2X messages with `speed` greater than $30km/h$ from the last minute every 10 seconds. We realize this by defining a time-based sliding window $W_{l_T}, v$ of size $l_T = 60s$ and with a sliding step of $v = 10s$ over the `CoCarMessage` stream. In Figure 5a the content of the window after 1 minute is shown. This is the first point in time, where the query with the window delivers results (hence, $\tau_i \geq \tau_0 + l_T$). The window contains the elements 1,2,4,5,6. After 10 more seconds the window slides, element 1 is dropped from the window and element 7 is added (Figure 5b). After another slide after 10 seconds element 2 is dropped and element 8 is added to the window (Figure 5c).




**(a)** Window after 60s **(b)** Window after 70s

**(c)** Window after 80s

**Figure 5** Example of a sliding window with size of $1min$ and slide step of $10s$.

Depending on the language, windows can either be implicitly included in an operator (see the definition of the Aggregate operator in Example 2, Figure 4a) or defined as a separate operator in the query language. An example of the latter is given in the data and query

model SStream [41] of the system SStreaMWare [42]. Gürgen et al. enable to define windows a little bit different to provide a high flexibility and variety of window types. A window creation operator produces a set of windows on a stream according to a window description configuring the operator. The description is constituted of initial start and end parameters (time or tuples are possible), parameters for the advancement of start and end window edges, and a rate parameter which is analog to the slide parameter. The windows serve as input for windowed operators, such as aggregation operators.

In SQL-based declarative languages the window construct included in the SQL:2003 standard is extended, e.g., by a `SLIDE` keyword to enable the definition of a sliding step. Example 11 shows three different types of windows in different languages.

▶ **Example 11.**

```
CREATE STREAM C2XSpeeder
SELECT COUNT(*) As speederNo
OVER (RANGE 1 MINUTE PRECEDING SLIDE 10 SECOND]
FROM C2XMessage WHERE Speed > 30.0
```

**Listing 2** A Time-based Sliding Window in ESL (Stream Mill)

```
SELECT Istream COUNT(*)
FROM C2XMessage[RANGE 100 SLIDE 100]
WHERE Speed > 30.0
```

**Listing 3** A Tuple-based Tumbling Window in CQL (STREAM)

```
SELECT Count(*)
FROM C2XMessage <LANDMARK RESET AFTER 600 ROWS ADVANCE 20 ROWS>
WHERE Speed > 30.0
```

**Listing 4** A Tuple-based Landmark Window in TruSQL (TruSQLEngine)

## 3.2 The Notions of Time and Order

We already mentioned before, that time plays an important role in DSMS. In all DSMS systems the processed tuples have some kind of timestamp assigned from a discrete and monotonic time domain. The timestamps allow then to determine if a tuple is in order or not and enable the definition of time-based windows [64].

### 3.2.1 Time

The prominent status of timestamps can already be seen from several semantics' definitions of streams. The timestamp is always handled as a specific attribute which is not part of the stream schema [57, 10, 50]. A monotonic time domain $T$ can be defined as an ordered, infinite set of discrete time instants $\tau \in T$ [57, 9]. For each timestamp exists a finite number of tuples (but it can also be zero).

In the literature, there exist several ways to distinguish where, when, and how timestamps are assigned. First of all, the temporal domain from which the timestamps are drawn can be either a *logical time domain* or *physical clock-time domain*. Logical timestamps can be simple consecutive integers, which do not contain any date or time information, but serve just for ordering. In contrast, physical clock-time includes time information (e.g., using UNIX timestamps). Furthermore, systems differ in which timestamps they accept

and use for internal processing (ordering and windowing). In most of the systems *implicit timestamps* [11, 78], also called *internal timestamps* or *system timestamps* are supported. Implicit timestamps are assigned to a tuple, when it arrives at the DSMS. This guarantees that tuples are already ordered by arrival time, when they are pipelined through the system. Implicit timestamps assigned at arrival in the system also allow for estimating the timeliness of the tuple when it is output [3]. Besides a global implicit timestamp (assigned on arrival), there exists also the concept of new (*local*) timestamps assigned at the input or output queue of each operator (time of tuple creation). This could also be implicitly expressed by the tuple's position in the queue [78]. In contrast, *explicit timestamps* [11, 78], *external timestamps* [15] or *application timestamps* [64] are created by the data stream sources and an attribute of the stream schema is determined to be the timestamp attribute. The authors of the Stream Mill language ESL [72, 15] use the term *explicit timestamp* in another way to discriminate between their concept of *latent timestamps* and internal and external timestamps. Latent timestamps are assigned on demand (lazily), i.e., only for operations dependent on a timestamp such as windowed aggregates [15], while explicit timestamps are assigned to every tuple. Example 12 shows, how the three types of timestamps are defined on creation of our C2XMsgs stream. In Listing 5 the creation timestamp `ts` is used as an explicit timestamp, designated by the `ORDER BY` clause. The query in Listing 6 uses an implicit timestamp by applying the function `current_time` and also designating it as order criterion. Listing 7 does not contain any information about an ordering attribute (no `ORDER BY` clause). Thus a timestamp will be assigned on demand.

▶ **Example 12** (Usage of the different timestamp types in ESL).

```
CREATE STREAM C2XMgs(
        ts timestamp, msgID char(10), lng real,
        lat real, speed real, accel real)
ORDER BY ts;
SOURCE 'port5678';
```

**Listing 5** Explicit Timestamp

```
CREATE STREAM C2XMgs(
        ts timestamp, msgID char(10), lng real, lat real,
        speed real, accel real, current_time
        timestamp)
ORDER BY current_time;
SOURCE 'port5678';
```

**Listing 6** Implicit Timestamp

```
CREATE STREAM C2XMgs(
        ts timestamp, msgID char(10), lng real, lat real,
        speed real, accel real);
SOURCE 'port5678';
```

**Listing 7** Latent Timestamp

Depending on the semantics of a data stream, tuples can include more than one timestamp. For example, in the data stream definition of the CESAR language [26], which is an event-based stream algebra, a tuple contains two timestamps, $\tau_0$ and $\tau_1$, denoting the start and end of an event, respectively.

There are also systems (e.g., StreamBase) which additionally order the tuples for the same timestamp, for instance, by arrival order. Jain et al. define the order of tuples with respect to the timestamp even stricter – an additional ordering between batches of tuples with the same timestamp is specified [45]. This order was proposed because the authors noticed semantic inconsistencies for query results in systems which use a *time-driven model* (i.e., windows are evaluated on each new time instant) as well as in systems, which use a *tuple-driven model* (i.e., a window is evaluated on each new tuple).

An interesting question is how timestamps should be assigned to results of binary operators and aggregates to ensure semantic correctness. Babcock et al. propose two solutions to assign a timestamp to results of a join [11]. The first option is to use the creation time of a join output tuple when using an implicit timestamp model. The second option is to use the timestamp of the first table involved in the join in the FROM clause of the query can be used, which is suited for explicit and implicit timestamp models. For aggregates similar considerations can be made. For example, if a continuous or windowed minimum or maximum is calculated, the timestamp of the maximal or minimal tuple, respectively, could be used. When a continuous sum or count is calculated, the creation time of the result tuple or the timestamp of the latest element included in the result can be used. If an aggregate is windowed there exist additional possibilities. The smallest or the highest timestamp of the elements in the window can be used as they reflect the oldest timestamp or most recent timestamp in the window, respectively. Both maybe interesting, when timeliness for an output tuple is calculated, but which one to use depends obviously on the desired outcome. Another possibility would be to take the median timestamp of the window.

### 3.2.2 Order

Many of the systems and their operators rely on (and assume) the ordered arrival of tuples in increasing timestamp order to be semantically correct [64]. For example, in the STREAM system (using a time-driven execution model) time can only advance to the next time instant, when all elements in the current time instant have been processed [10]. This has been coined as the *ordering requirement* [64]. But as already pointed out, this can not be guaranteed especially for explicit timestamps and data from multiple sources. In the various DSMS basically two main approaches to the problem of disorder have been proposed.

One approach is to tolerate disorder in controlled bounds. The Aurora system, for example, does not assume tuples to be ordered by timestamp [3]. The system divides operators into *order-agnostic* and *order-sensitive* operators. The first group of operators does not rely on an ordering of elements, for instance, the Filter operation we already introduced in Example 2, which is a unary operation processing one tuple at a time. The order-sensitive operators are parametrized with a definition how unordered tuples should be handled. The definition contains the attribute which indicates the order of the tuples and a *slack* parameter. The slack parameter denotes, how many out-of-order tuples may arrive between the last and next in-order tuple. All further out-of-order tuples will be discarded. The order can also be checked for partitions of tuples, specified by an additional GROUP BY clause in the order definition. A general concept of a slack parameter, called *adherence parameter* has been presented for the STREAM system [7, 13]. The adherence parameter is a measure for how well a stream "adheres" to a defined constraint. The authors define a set of *k-constraints* one of which is the ordered-arrival-k-constraint. This constraint conforms to the slack parameter's ordering semantics.

The second way to handle disorder is to dictate the order of tuples and reorder them if necessary. While the use of implicit timestamps is a simple way of ordering tuples on

arrival [64], the application semantics often requires the use of explicit timestamps though. *Heartbeats* [64] are tuples sent with the stream including at least a timestamp. These markers indicate to the processing operators, that all following tuples have to have a timestamp greater than the timestamp in the punctuation. As already mentioned in Section 2 the STREAM system includes an Input Manager which buffers stream elements based on heartbeats. The buffered elements are output in ascending order as soon as a heartbeat is received, i.e., they are locally sorted. The heartbeats can be created by the stream sources or by the Input Manager itself. Srivastava and Widom propose approaches to create heartbeats either periodically or based on properties of the stream sources and the network, such as transmission delay [64]. Heartbeats are only one possible form of *punctuation* [74]. Punctuations, in general, can contain arbitrary patterns which have to be evaluated by operators to true or false [74]. Therefore, punctuations can also be used for approximation. They can limit the evaluation time or the number of tuples which are processed by an otherwise blocking or stateful operator. For example, when a punctuation-aware join operator receives a punctuation it can match all elements received on the streams to join since the last punctuation. Other methods for reordering tuples in limited bounds use specific operators. Aurora's SQuAl language provides a sorting operator called BSort [3]. It orders tuples according to some attribute by applying a buffered bubble sort algorithm. Finally, the Stream Mill system leaves the handling of out-of-order tuples to the user [15]. It detects these tuples and adds them to a separate stream.

## 3.3 Completeness of Stream Query Languages

It has already been mentioned that blocking operators are not an option for query languages in DSMS [11]. Hence, SQL as is is not suited for DSMS, because sequence queries cannot be expressed [52, 51]. An interesting question therefore is, which queries can be formulated using only non-blocking operators of SQL. Continuous query semantics is based on the append-only principle. Therefore, the class of monotonic queries is different from the query classes which allow deletions and updates [70]. For a monotonic query $Q$ holds, that the result of a query over the ordered stream $S$ at time $\tau_i$ is included in the results of the query at time $\tau_{i+1}$ [70, 52, 51, 39] or formally expressed:

$$Q(S(\tau_i)) \subseteq Q(S(\tau_{i+1})), \forall \tau_i \in \mathbb{T}$$

Law et al. have proven that the class of monotonic queries over data streams can be expressed by queries using only non-blocking operators [52]. Non-blocking and monotonic operators in the relational algebra are obviously Selection and Projection. Law et al. also showed that a query which is monotonic for ordered streams is also monotonic for relations with respect to set containment and can therefore be expressed only with non-blocking operators [51]. It can be followed from this, that Union and Cartesian Product or Join also can be calculated by non-blocking operators as these are monotonic wrt. set containment [51], while Set Difference (which is non-monotonic) cannot. The intersection is a monotonic operator and can be either expressed in the relational algebra by non-blocking or blocking operators. In SQL also continuous forms of Count and Sum are non-blocking [52].

# 4 Data Quality in DSMS

We have motivated the demand for a new concept of data management systems by monitoring and tracking applications. These applications usually rely on sensors which create data streams by measuring values or recording multimedia. But the use of sensors also reveals

problems. The produced data can be incorrect, unordered, and incomplete. Another source of inaccuracies in data streams can be the use of classification, prediction, ranking or any other sort of approximation algorithms [47, 48]. The unreliability of the data processed again leads to unreliable results of applications realized with a DSMS. Consider our example of traffic state estimation. We rely on positions determined by GPS devices. These devices usually introduce a measurement error in positioning and messages may also contain no position, when the GPS signal is lost. These errors are propagated through the entire data processing chain. For example, when the road and the section on which the message has been created is determined, the error can lead to an assignment to a wrong section. Inevitably, this will lead to inexact results when estimating the traffic state for the road. Hence, means to take inaccuracies into account and to rate the result of a query have to be considered. Not only the quality of the data values (we will call this *application-based data quality* in the following), but also the data stream management system performance has to be considered to assess the quality of query results. When the output delay in the DSMS is too high, e.g., the estimated traffic state will identify a traffic situation which is no longer present. Furthermore, if too many tuples are dropped by a load shedder to gain performance, statements based on a low number of data may also harm the result. Therefore, also the Quality of Service (QoS) for multiple performance aspects of a system has to be tracked and taken into account in query processing. Hence, we define data quality related to the system performance as *system-based data quality*. In the following, we will discuss data quality dimensions for application-based and system-based data quality. We will discuss solutions which enable to rate and to track the corresponding data quality dimensions. We will also briefly introduce our own ontology-based approach.

## 4.1 Application-based and System-based Data Quality Dimensions

As mentioned before, we distinguish data quality which is inherent in the data itself, i.e., it describes how reliable the data we process is, and data quality which represents the system performance of a DSMS. The type of the measured data quality is expressed by a *data quality dimension*. For example, the timeliness of data is a data quality dimension. For each data quality dimension a *data quality metric* defines a possible way to measure the quality within that dimension. There exists a plethora of classifications which structure and describe data quality dimensions, such as the Total Data Quality Management (TDQM) classification [75, 66], the Redman classification [60], or the Data Warehouse Quality (DWQ) classification [46]. For data streams, Klein and Lehner propose a dimension classification [49]. In Table 1 we list a non-exhaustive set of data quality dimensions, which we think are of importance for data quality rating in a traffic state estimation application realized by a DSMS (based on [49, 16]) and rate if they are application-based or system-based. Data quality for a dimension can be measured on different levels. It can be measured system-wide, e.g., the output rate, on operator level, e.g., the selectivity of an operator, or on window, tuple, or attribute level.

In the following, we will discuss some approaches of measuring and rating data quality in DSMS.

## 4.2 Quality of Service Monitoring in DSMS

In Section 2 we described that DSMS can implement means to monitor the system performance during query processing. Aspects which describe the performance of a DSMS are also termed *Quality of Service*. Quality of Service in general rates how good the component or system at

**Table 1** Example Data Quality Dimensions.

| Data Quality Dimension | Informal Description | Example Metric | Application-based/ System-based |
|---|---|---|---|
| Completeness | Ratio of missing values or tuples to the number of received values/tuples | The number of non-null values divided by all values including null values in a window | Application-based |
| Data Volume | The number of tuples or values a result is based on, e.g., the number of tuples used to calculate an aggregation | Quantity of tuples in a window | System-based and Application-based |
| Timeliness | The age of a tuple or value | Difference between creation time and current system time | System-based and Application-based |
| Accuracy | Indicates the accuracy of the data, e.g., a constant measurement error or the result of a data mining algorithm | An externally calculated or set value | Application-based |
| Consistency | Indicates the degree to which a value of an attribute adheres to defined constraints, e.g., if a value lies in certain bounds | Rule evaluation | Application-based |
| Confidence | Reliability of a value or tuple, e.g., the confidence to have estimated the correct traffic state | A weighted formula which is calculated from values for other data quality dimensions | Application-based |

hand fulfils the constraints and requirements posed to it. QoS oriented systems then try to give guarantees for the QoS aspects and try to keep QoS within defined bounds by applying countermeasures. QoS dimensions (used here analogously to quality dimensions) have been classified by Schmidt in [62] into time-based and content-based dimensions. Schmidt identified the following time-based dimensions for DSMS: throughput (or data rate), output delay (or latency) and the following content-based dimensions: sampling (or drop rate), sliding window size, approximation quality and data mining quality. He defined two new time-based dimensions, called signal frequency (amount of information in a stream) and inconsistency (maximal difference between creation timestamp and system timestamp, which is similar to our example metric of timeliness in Section 4.1). In the Aurora system additionally a value-based QoS dimension is defined, which rates if important values have been output, i.e., they prioritize results and can therefore also prioritize values and adapt operators of the corresponding queries for these values [3]. To guarantee to stay in given bounds for the above dimensions, DSMS have several countermeasures, depending on the QoS dimension at hand.

For example, if the system is overloaded and the output delay or the throughput are low, a DSMS can drop tuples with sampling techniques, which can be quite simple (random) or very sophisticated, e.g., based on information about other QoS parameters as in the case of the value-based QoS [3]. The dropping of tuples is also called *load shedding*. Load shedding is done by placing load shedding operators into the query execution plan [12, 69] where required. Other possibilities are adaptive load distribution or admission control [69]. In the Aurora system for each QoS the administrator of the system is required to model a two-dimensional function with a QoS rating between 0 and 1 on the y-axis and the QoS dimension values on the x-axis for each output stream [3]. Additionally, a threshold for the QoS dimension has to be defined to indicate in which bounds QoS is acceptable. A similar approach is followed by the QStream system [61, 62]. In the QStream system two descriptors for each output stream are defined – a content quality descriptor which includes the dimension's inconsistency and signal frequency defined by Schmidt [62] and a time quality descriptor consisting of values for data rate and delay. The quality dimensions are calculated throughout the whole query process. All operators in the query execution plan comprise functions used to calculate the value of each quality dimension when new tuples are processed. At the end of the processing chain quality values are output for the result streams of each continuous query. A user can pose requirements on the result stream's quality by formulating a request describing thresholds for the dimensions. If the descriptors meet the quality request, this is reported to the user as a successful negotiation.

## 4.3 Inclusion of Data Quality in Data Streams

In the systems reviewed in the previous section, QoS information is handled separately from the stream data and can only be retrieved for the output streams. Furthermore, data quality dimensions are mostly system-based, i.e., the content of the data in the stream and corresponding application-based quality dimensions are not taken into account. In the successor system of Aurora, Borealis [1], the QoS model of Aurora has been refined to rate QoS not at the output streams of the system, but also in each operator in between. To this end, each tuple includes a vector with quality dimensions, which can be content-related (e.g., the importance of a message) or performance-related (e.g., processing time for a message up to this operator). The vectors can be updated by operators in the query execution plan and a score function is provided which can rate the influence of a tuple on the QoS based on a vector [1]. A crucial limitation of the previous approach is, that the quality dimensions in the vector are equal for each stream, which does not allow for an application-based data quality rating of stream contents. To achieve a more fine-granular rating of data quality on attribute, tuple and window level and to also include application-based data quality dimensions, Klein and Lehner [49] propose a different approach. They extended the PIPES system [50] by Krämer and Seeger with modified operators to include data quality dimensions as a part of the stream schema. Krämer and Seeger distinguish four different types of operators based on the operator's influence on the stream data (modifying, generating, reducing or merging operators). Changes on the data can in turn result in updates of the data quality of an attribute, tuple, or window. For each data quality dimension and each operator the influence of the operator is discussed and a function to calculate the new data quality is provided. The approach introduces so called *jumping data quality windows* which include a set of data quality dimensions and where for each attribute and window the size of the window can be defined independently [49]. In addition, Klein and Lehner make the size of the data quality windows dynamically adaptable based on an *interestingness factor*. The interestingness factor is dependent on the application realized and can for example shrink the window size when

there are interesting peaks in the stream data to refine the granularity of quality information in this stream portion.

A drawback of Klein and Lehner is the deep integration of data quality dimensions and corresponding metrics into the operators. The implementation of operators has to be changed substantially to include the quality information. Therefore, we propose a more flexible solution, which allows to define custom quality dimensions and metrics based on an abstract data quality ontology. The ontology consists of two parts – a first part including data quality related concepts, such as Quality Metric, Quality Dimension and Quality Factor, and a second part comprising application specific concepts with relationships to the data quality concepts (e.g., an attribute is related to a certain quality dimension). The data quality meta information is loaded once when a continuous query is registered in the system and the data quality is then calculated according to the meta information throughout query processing. Similar to [49] we add the data quality information as separate attributes to the tuples in the stream. We identified three main tasks in the data stream management process, which can influence the data quality. First, in line with Klein and Lehner we parse the continuous query and identify relational operators in the query which modify the data stream and the data in it and hence, also the data quality. Second, in the ontology we allow to define rules, which rate the semantic consistency of attribute values or values of multiple related attributes in a tuple. For example, if we receive the current temperature and an indicator for snow, we can define a rule to check if the value for snow (yes/no) is consistent with the temperature given (no snow possible when above $3°C$). Third, also application specific and user-defined operations, such as data mining operations, are considered and the addition of data quality values can be easily integrated into the application code. The data quality metrics can be arbitrarily complex and can include several data quality factors and can also combine values of other data quality dimensions. This allows for flexible and application-dependent estimation of data quality. For example, in the traffic state estimation example we calculate a confidence value for the estimated traffic state based on multiple weighted data quality dimensions, where the weights allow to reflect the individual contribution to the overall quality. Finally, we made the calculation of data quality in the system optional – if not required, data quality information is not calculated at all. Our approach is described in detail in [36].

## 4.4 Probabilistic Data Streams

A completely different way of dealing with defective data in DSMS are probabilistic data streams. Uncertainty has been studied intensively for Database Management Systems and several systems (mainly research prototypes) implementing probabilistic query processing have been proposed. In data stream management this is a very recent topic which has been addressed only scarcely in literature compared to other research questions. Kanagal and Desphande [48] distinguished two main types of uncertainties in data streams (analogously to databases). First, the existence of a tuple in the stream can be uncertain (how probable is it for this tuple to be present at the current time instant?), which is termed by Kanagal and Despande *tuple existence uncertainty*. Second, the value of an attribute in a stream tuple can be uncertain, which is called *attribute value uncertainty* (what is the probability of attribute X to have a certain value?). In literature, mainly the tuple existence uncertainty is addressed. To model uncertainty for attribute values, for each attribute of a stream a random variable with a corresponding distribution function has to be introduced [48]. The probability of a tuple to consist of a certain configuration of values is then modeled by a joint distribution function, which multiplies the probabilities of the single attribute values. The tuple existence uncertainty can be modeled by a binary random variable, which can either be zero (is not included) or one (is included) and a probability distribution function [48].

▶ **Definition 13.** A *probabilistic stream* or *probabilistic sequence* is a sequence of tuples $S = < t_1, p_1 >, \ldots, < t_n, p_n >$, where $p_i$ is the probability of the existence of a tuple $t_i$ at position $i$ modeled by a probability distribution function (pdf) [48, 47, 23].

The probability for a composition of the stream of a certain multiset of tuples at a certain point in time is again modeled by a joint probability distribution calculated from the pdfs of the tuples. Analogously to the possible worlds in probabilistic databases, from the random variables and the probability distributions, all *possible streams* [48, 47], i.e., deterministic instances of the stream, at a time $\tau$ and their probability can be determined.

One can easily see that the number of possible streams grows exponentially with the size of the stream and the number of possible tuples (#P-hard data complexity) [23]. For certain query operations, such as aggregates, on probabilistic streams, which are again probability distributions, it is therefore undesirable to calculate all the possible streams. The remedy to this problem are operators, which approximate the pdfs for the corresponding query operations. In [23] aggregates are computed using sketches, i.e., estimating frequency moments for the probabilistic stream. The data complexity of possible streams is even worse when the random variables are not independent of each other, i.e., if there is a correlation between tuples [48]. Kanagal and Desphande [48] propose to use directed graphical models (this can be, for example, Bayesian Networks) to describe correlated random variables, intermediate query results, and the dependencies between them. In the graphical model, the nodes depict the random variables and the edges are the dependencies. To limit the size of the graph and hence the computational complexity for a query, marginal distributions are used, which neglect unnecessary dependencies and infer new dependencies. Kanagal and Desphande extended the set of SQL operators by some probabilistic operators. For example, two new `SELECT` operators are provided, which can return a deterministic result for a query. The `SELECT-MAP` retrieves the stream with the highest probability of all possible streams, while `SELECT-ML` returns the possible stream which includes for each attribute of a tuple the value with the highest probability for that attribute. Their language also supports sliding windows and the set of common aggregates over the probabilistic data streams.

While the approaches before assumed possible stream semantics for streams with discrete data, these semantics is not valid for streams with uncertain continuous data (a problem of attribute-value uncertainty). The PODS system [73] addresses this issue by modeling each continuous-valued attribute as a continuous random variable with a corresponding pdf. To model pdfs for continuous random variables Tran et al. use Gaussian Mixture Models, which include several distributions for one variable. A bimodal distribution function, for example, can be approximated by two different Gaussian distribution functions. A probabilistic or uncertain data stream then consists of a sequence of tuples with discrete and continuous random variables for each attribute [73].

## 5 Conclusion

DSMS have demonstrated to be effective and efficient solutions dealing with huge amounts of rapidly incoming data which has to be processed in real-time. The new data management requirements of data stream applications have not only led to new concepts for data management architectures, but also to new and adapted query languages and semantics suited for streaming systems. Now, after almost two decades of research on continuous queries and data streams, many research prototypes have evolved and already some of them have turned into mature industry solutions. Many big players in the field of data management, such as Microsoft, IBM, or Oracle, have their own data stream management solution. However, despite

some efforts, there is neither a common standard for query languages nor an agreement on a common set of operators and their semantics until today. Finally, data quality has turned out to be a crucial aspect in DSMS. Incoming data streams may be delivered by unreliable sources and results of a DSMS application can be falsified by data inaccuracies. Hence, data quality measurement, monitoring, and improvement solutions have to be integrated into DSMS and receive increasing attention. Since many interesting questions still remain open further research in the field is to be done and will follow.

**Acknowledgments.** I thank my colleagues Stefan Schiffer and Christoph Quix and the reviewers for their valuable comments.

## References

1 Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. 2nd Biennal Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, Asilomar, CA, USA, 2005.

2 Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A data stream management system. In Halevy et al. [44], page 666.

3 Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

4 Karl Aberer, Manfred Hauswirth, and Ali Salehi. A middleware for fast and flexible sensor network deployment. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proc. 32nd Intl. Conference on Very Large Data Bases (VLDB)*, pages 1199–1202. ACM Press, 2006.

5 Yanif Ahmad and Ugur Çetintemel. Data Stream Management Architectures and Prototypes. In Liu and Özsu [54], chapter D, pages 639–643.

6 A. Angel, N. Sarkas, N. Koudas, and D. Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *Proceedings of the VLDB Endowment*, 5(6):574–585, 2012.

7 Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. Technical report, Stanford InfoLab, 2004.

8 Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. In Halevy et al. [44], page 665.

9 Arvind Arasu, Shivnath Babu, and Jennifer Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. In *Proc. Intl. Conf. on Data Base Programming Languages*, 2003.

10 Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

11 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In Lucian Popa, editor, *Proc. 21st ACM Symposium on Principles of Database Systems (PODS)*, pages 1–16, Madison, Wisconsin, 2002. ACM Press.

12 Brian Babcock, Mayur Datar, and Rajeev Motwani. Load Shedding in Data Stream Systems. In Charu Aggarwal, editor, *Data Streams - Models and Algorithms*, chapter 7, pages 127–147. Springer, 2007.

13 S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.

14 Shivntah Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–119, 2001.

15 Yijian Bai, Richard Chang Luo, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. *An Introduction to the Expressive Stream Language (ESL)*. UCLA, CS Department, 2004.

16 Norbert Baumgartner, Wolfgang Gottesheim, Stefan Mitsch, Werner Retschitzegger, and Wieland Schwinger. Improving situation awareness in traffic management. In *Proc. of the 30th Intl. Conf. on Very Large Databases*, 2010.

17 A. Bolles. A flexible framework for multisensor data fusion using data stream management technologies. In *Proceedings of the 2009 EDBT/ICDT PhD Workshop*, pages 193–200. ACM, 2009.

18 I. Botan, D. Kossmann, P.M. Fischer, T. Kraska, D. Florescu, and R. Tamosevicius. Extending XQuery with window functions. In *Proceedings of the 33rd international conference on Very large data bases*, pages 75–86. VLDB Endowment, 2007.

19 Don Carney, Ugur Centintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams - a new class of data management applications. In *Proc. 28th Intl. Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002. Morgan Kaufmann.

20 Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing for an uncertain world. In *Proc. 1st Biennal Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 2003.

21 Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–390, Dallas, Texas, 2000. ACM.

22 Mitch Cherniack and Stan Zdonik. Stream-oriented query languages and architectures. In Liu and Özsu [54], chapter S, pages 2848–2854.

23 Graham Cormode and Minos Garofalakis. Sketching probabilistic data streams. In Lizhu Zhou, Tok Wang Ling, and Beng Chin Ooi, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Beijing, China, 2007. ACM Press.

24 Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In Halevy et al. [44], pages 647–651.

25 G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

26 Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. A general algebra and implementation for monitoring event streams. Technical report, Cornell University, 2005. http://hdl.handle.net/1813/5697.

27 Ramez A. Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 4th edition, 2004.

28 Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 2011.

**29** European Union. Uniform provisions concerning the approval of vehicles with regard to the speedometer equipment including its installation. *Official Journal of the European Union*, 53:40–48, May 2010.

**30** J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, 2005.

**31** Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

**32** D. Geesen and M. Grawunder. Odysseus as platform to solve grand challenges: Debs grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 359–364. ACM, 2012.

**33** Sandra Geisler, Yuan Chen, Christoph Quix, and Guido G. Gehlen. Accuracy Assessment for Traffic Information Derived from Floating Phone Data. In *Proc. of the 17th World Congress on Intelligent Transportation Systems and Services*, 2010.

**34** Sandra Geisler, Christoph Quix, and Stefan Schiffer. A data stream-based evaluation framework for traffic information systems. In Mohamed Ali, Erik Hoel, and Cyrus Shahabi, editors, *Proc. of the 1st ACM SIGSPATIAL International Workshop on GeoStreaming*, pages 11–18, November 2010.

**35** Sandra Geisler, Christoph Quix, Stefan Schiffer, and Matthias Jarke. An evaluation framework for traffic information systems based on data streams. *Transportation Research Part C*, 23:29–55, August 2012.

**36** Sandra Geisler, Sven Weber, and Christoph Quix. An ontology-based data quality framework for data stream applications. In *Proc. 16th Intl. Conf. on Information Quality (ICIQ)*, Adelaide, Australia, 2011.

**37** T.M. Ghanem, W.G. Aref, and A.K. Elmagarmid. Exploiting predicate-window semantics over data streams. *ACM SIGMOD Record*, 35(1):3–8, 2006.

**38** Lukasz Golab and M. Tamer Özsu. Issues in stream management. *SIGMOD Record*, 32:5–14, 2003.

**39** Lukasz Golab and M. Tamer Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

**40** Y. Gurevich, D. Leinders, and J. Van den Bussche. A theory of stream queries. In *Proceedings of the 11th international conference on Database programming languages*, pages 153–168. Springer-Verlag, 2007.

**41** L. Gürgen, C. Labbé, C. Roncancio, and V. Olive. Sstream: A model for representing sensor data and sensor queries. In *Int. Conf. on Intelligent Systems And Computing: Theory And Applications (ISYC)*, 2006.

**42** L. Gürgen, C. Roncancio, C. Labbé, A. Bottaro, and V. Olive. Sstreamware: a service oriented middleware for heterogeneous sensor data management. In *Proceedings of the 5th international conference on Pervasive services*, pages 121–130. ACM, 2008.

**43** Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. SASE: Complex Event Processing Over Streams. In *Proc. of 3rd Biennal Conference on Innovative Data Systems Research (CIDR)*, 2007.

**44** Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors. *Proc. ACM SIGMOD Intl. Conference on Management of Data*, San Diego, California, USA, 2003. ACM.

**45** Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1379–1390, 2008.

**46** Matthias Jarke, Manfred A. Jeusfeld, C. Quix, and Panos Vassiliadis. Architecture and Quality in Data Warehouses: An Extended Repository Approach. *Information Systems*, 24(3):229–253, 1999.

**47** TS Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 346–355. Society for Industrial and Applied Mathematics, 2007.

**48** Bhargav Kanagal and Amol Deshpande. Efficient query evaluation over temporally correlated probabilistic streams. In *Proc. of IEEE Intl. Conf. on Data Engineering*, 2009.

**49** A. Klein and W. Lehner. Representing data quality in sensor data streaming environments. *ACM Journal of Data and Information Quality*, 1(2):1–28, September 2009.

**50** Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continous sliding window queries over data streams. *ACM Trans. on Database Systems*, 34:1–49, 2009.

**51** Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Relational Languages and Data Models for Continuous Queries on Sequences and Data Streams. *ACM Transactions on Embedded Computing Systems*, 36(3):1–31, March 2011.

**52** Y.N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proceedings of the 30th Intl. Conf. on Very large data bases*, pages 492–503. VLDB Endowment, 2004.

**53** J. Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM, 2005.

**54** Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer, 2009.

**55** Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):610–628, 1999.

**56** David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of data streams and operators. In *ICDT 2005*, number 3363 in LNCS, pages 37–52. Springer, 2005.

**57** Kostas Patroumpas and Timos K. Sellis. Window specification over data streams. In *Current Trends in Database Technology - EDBT 2006 Workshops*, pages 445–464, 2006.

**58** Feng Peng and Sudarshan S. Chawathe. Xsq: A streaming xpath engine. Technical report, Computer Science Department, University of Maryland, 2003.

**59** Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 353–366, Bangalore, India, 2003. IEEE Computer Society.

**60** Thomas C. Redman. *Data Quality for the Information Age*. Artech House, Boston, 1996.

**61** S. Schmidt, H. Berthold, and W. Lehner. Qstream: Deterministic querying of data streams. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proc. 30th Intl. Conference on Very Large Data Bases (VLDB)*, pages 1365–1368, Toronto, Canada, 2004. Morgan Kaufmann.

**62** Sven Schmidt. *Quality-of-Service-Aware Data Stream Processing*. PhD thesis, Technischen Universität Dresden, 2006.

**63** D. Singh, A. Ibrahim, T. Yohanna, and J. Singh. An overview of the applications of multisets. *Novi Sad Journal of Mathematics*, 37(3):73–92, 2007.

**64** Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In Alin Deutsch, editor, *Proc. 23rd ACM Symposium on Principles of Database Systems (PODS)*, pages 263–274, Paris, France, 2004. ACM.

**65** Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

**66** D.M. Strong, Y.W. Lee, and R.Y. Wang. Data quality in context. *Communications of the ACM*, 40(5):103–110, 1997.

**67** J. Sun, C. Faloutsos, S. Papadimitriou, and P.S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 687–696. ACM, 2007.

**68** Apostolos Syropoulos. Mathematics of multisets. In *Proceedings of the Workshop on Multiset Processing*, pages 286–295, 2000.

**69** N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

**70** Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *ACM SIGMOD 1992*, 1992.

**71** Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. Continuous queries over append-only databases. In Michael Stonebraker, editor, *Proc. ACM SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, CA, 1992. ACM Press.

**72** H. Thakkar, B. Mozafari, and C. Zaniolo. Designing an inductive data stream management system: the stream mill experience. In *Proceedings of the 2nd international workshop on Scalable stream processing system*, pages 79–88. ACM, 2008.

**73** T.T.L. Tran, L. Peng, B. Li, Y. Diao, and A. Liu. PODS: a new model and processing algorithms for uncertain data streams. In *Proceedings of the 2010 international conference on Management of data*, pages 159–170. ACM, 2010.

**74** P.A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, pages 555–568, 2003.

**75** R.Y. Wang and D.M. Strong. Beyond accuracy: What data quality means to data consumers. *Journal of management information systems*, 12(4):5–33, 1996.

**76** J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292–2330, 2008.

**77** C. Zaniolo. Logical foundations of continuous query languages for data streams. *Datalog in Academia and Industry*, pages 177–189, 2012.

**78** Stan Zdonik, Peter Sibley, Alexander Rasin, Victoria Sweetser, Philip Montgomery, Jenine Turner, John Wicks, Alexander Zgolinski, Derek Snyder, Mark Humphrey, and Charles Williamson. Streaming for dummies. http://list.cs.brown.edu/courses/csci2270/archives/2004/papers/paper.pdf, 2004.

# A Scientific and Commercial DSMS

In the following tables scientific and commercial DSMS are listed with their predominant properties.

## A.1 Research Projects

**Table 2** Data Stream Management Systems – Research Projects.

| Project | Research Group | Runtime | Description |
|---|---|---|---|
| Tapestry [71] | Xerox Parc (Terry, Goldberg et al.) | 1992–? | Uses a commercial append-only database, cont. querying by using stored procedures |
| NiagaraCQ [21] (http://research.cs.wisc.edu/niagara) | University of Wisconsin-Madison (Chen, DeWitt et al.) | 2000–2002 | Distributed ystem for continuous queries over web resources in XML format. Queries are executed periodically after a fixed amount of time or on arrival of new data. Windows are not supported. Language: XML-QL |
| Gigascope [24] | AT&T Labs and CMU (Cranor, Johnson, Srivastava et al.) | Ongoing | Specifically conceptualized for network monitoring applications. Language: GSQL |
| OpenCQ [55] (http://www.cc.gatech.edu/projects/disl/CQ) | College of Computing at the Georgia Institute of Technology (Liu, Pu et al.) | 1999–2002 | System to execute continuous queries on a trigger basis, i.e., when a trigger condition is fulfilled (database modifications, time events, or user-defined events), a query is executed. |
| TelegraphCQ [20] (http://telegraph.cs.berkeley.edu) | UC Berkeley (Hellerstein, Franklin et al.) | 2000–2007 | Reuses components from DBMS PostgreSQL, dataflows composed of set of operators (e.g., Eddy, Join) connected by Fjords, Language: SQL, scripts |
| STREAM [7] (http://infolab.stanford.edu/stream) | Stanford University (A. Arasu, J. Widom, B. Babcock, S. Babu et al.) | 2000–2006 | Probably the most famous one, comprehensible abstract semantics description; Language: CQL |
| Aurora/Borealis [2, 1] (http://www.cs.brown.edu/research/borealis) | Brown Univ., Brandeis Univ., MIT (Abadi, Cherniack, Madden, Zdonik, Stonebraker et al.) | 2003–2008 | Distributed system, uses notions of arrows, boxes and connection points for operator networks; Commercial: StreamBase; Language SQuAl |
| PIPES [50] (http://dbs.mathematik.uni-marburg.de/Home/Research/Projects/PIPES) | Universität Marburg (Seeger, Krämer et al.) | 2003–2007 | Commercial: RTM Analyzer; Language: PIPES, define logical and physical query algebra on multisets, use algebraic optimizations |
| System S/SPC/SPADE [31] (http://researcher.watson.ibm.com/researcher/view_project.php?id=2531) | IBM T.J. Watson Research | Started 2006; Ongoing | Distributed System, notion of operator network, Commercial: InfoSphere Streams; Language: SPADE/SPL |
| StreamMill [72] (http://wis.cs.ucla.edu/wis/stream-mill/index.php) | UCLA (H. Takkhar, C. Zaniolo) | Ongoing | Inductive DSMS (mining implementable with SQL and UDAs), support for XML data;Language: ESL |
| Global Sensor Networks [4] (http://sourceforge.net/apps/trac/gsn/) | EPF Lausanne, Digital Enterprise Research Insitute (DERI) (Salehi, Aberer et al.) | Ongoing | Wraps existing rel. DBMS with stream functionality; language: SQL |
| Odysseus [32] (http://odysseus.offis.uni-oldenburg.de:8090/display/ODYSSEUS/Odysseus+Home) | Carl von Ossietzky Universität Oldenburg (A. Bolles, D. Geesen, M. Grawunder, J. Jacobi, D. Nicklas) | Ongoing | Framework to create custom DSMS by extending and adapting architectural components. |
| SStreaMWare [41, 42] | France Telecom and LIG Laboratory (L. Gürgen et al.) | 2006-2008 | Service-oriented framework focusing on sensor data processing; Relational with a generic schema for sensor measurements and meta-data. Language: SQL-based |

## A.2 Commercial Systems

**Table 3** Data Stream Management Systems – Commercial Products.

| System | Company | Based on | Description |
|---|---|---|---|
| InfoSphere Streams http://www-01.ibm.com/software/data/infosphere/streams/ | IBM | System S / SPADE / SPC | Current version: 3.0; Stand-alone product, supports only Linux, queries over structured and unstructured data sources; Language: SPL (successor of SPADE) |
| Oracle Event Processing (OEP) (http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html) | Oracle | – | Standalone product. Current version: 11gR1; Available for various OS; based on Java and XML, Eclipse development plugin available (also for visual stream graph composition); Languages: CQL/EPL (Event processing language) |
| StreamInsight http://msdn.microsoft.com/de-de/library/ee362541.aspx | Microsoft | – | Standalone product. Needs SQL Server 2012 product key. Current version: 2.0 Languages: .NET, LINQ |
| StreamBase http://www.streambase.com | StreamBase | Aurora/Borealis | Stand-alone products (Server, Studio, Adapters, LiveView,...); Language: StreamSQL |
| TruSQL Engine | Truviso | TelegraphCQ | has been acquired by Cisco and integrated into Cisco Prime, a network management software bundle; Language: StreaQL |
| Esper (Open Source) http://esper.codehaus.org | EsperTech | – | Available for .NET and Java, Standalone product; Language: EPL |
| SAP Sybase Event Stream Processor (http://www.sybase.com/products/financialservicessolutions/complex-event-processing) | Sybase | – | Current Version: 5.1;Language: CCL (Continuous Computation Language) |